NPS-CS-07-008

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

The Three Dimensions of Formal Validation and Verification of Reactive System Behaviors

By

D. Drusinsky, J.B. Michael, and M. Shing

August 2007

Approved for public release; distribution is unlimited

Prepared for: NASA IV&V Facility
100 University Drive
Fairmont, WV 26554

THIS PAGE INTENTIONALLY LEFT BLANK

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

Daniel T. Oliver	Leonard A. Ferrari
President	Provost

This report was prepared for the NASA IV&V Facility and funded by the NASA IV&V Facility.

This report was prepared by:

James Bret Michael
Professor of Computer Science and Electrical Engineering
Naval Postgraduate School

Reviewed by:	Released by:
Peter J. Denning, Chairman	Dan C. Boger
Department of Computer Science	Interim Associate Provost and
	Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE | *Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** August 2007	**3. REPORT TYPE AND DATES COVERED** Technical Report

4. TITLE AND SUBTITLE: Title (Mix case letters)
The Three Dimensions of Formal Validation and Verification of Reactive System Behaviors

5. FUNDING NUMBERS
NNG07LD01I

6. AUTHOR(S)
D. Drusinsky, J.B. Michael, and M. Shing

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER NPS-CS-07-008

9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)
NASA IV&V Facility, 100 University Drive, Fairmont, WV 26554

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this report are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

In-spite of three decades of software formal verification and validation (FV&V) research, there exists no ideal FV&V technique that works well for all FV&V concerns. That is, there is no one technique that enables (i) easy and correct construction of requirement specification of complex real-life properties, and (ii) complete verification coverage of complete real-life complex software with respect to those requirements. Moreover, many of the FV&V techniques are ineffective in handling temporal behavior of reactive systems.

In this paper we use a cuboid to characterize the trade space among three categories of FV&V techniques. We illustrate the use of the cuboid in tradeoff analysis to determine the appropriate techniques for V&V based on cost and coverage.

14. SUBJECT TERMS
Validation and verification, formal methods, model checking, runtime verification

15. NUMBER OF PAGES
22

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

THIS PAGE INTENTIONALLY LEFT BLANK

1. INTRODUCTION

When we type the word “software problem” into any Internet search engine, we can easily come up with dozens of articles reporting the impact of software problems in our lives, such as the malfunctioning of the Miele G885 SC dishwasher, worldwide recall of the BMW 745i sedan, the shut down of Southern California's airspace due to a software glitch leaving controllers without maps showing terrain and airspace boundaries on their radarscopes, the loss of an Ariane 5 rocket and its payload satellite, and the loss of life due to friendly fire by the Patriot missile defense system. Software is ubiquitous, and software errors affect everybody. A study sponsored by the National Institute of Standard and Technology (NIST) in 2001 found that the annual cost of software errors to the U.S. economy is approximately $59.5 billion, which is about 0.6 percent of the gross domestic product [1]. Moreover, some of these errors, particularly those in software-intensive reactive systems, may have catastrophic consequences.

Reactive systems (or subsystems) are systems that perform an ongoing and often never-ending computation, in which each invocation uses information generated by previous invocations.[1] Examples of reactive systems include all kinds of controllers. In contrast, transformational systems (or subsystems) are those in which the result of an invocation (call) does not depend on previous invocations, such as a square root method or a Fast Fourier Transform (FFT) method.

The activities for assuring the correctness of reactive systems reside within the Validation and Verification (V&V) process. According to the *Guide to the Software Engineering Body of Knowledge* [2],

> The V&V process determines whether or not products of a given development or maintenance activity conform to the requirement of that activity, and whether or not the final software product fulfills its intended purpose and meets user requirements. Verification is an attempt to ensure that the product is built correctly, in the sense that the output products of an activity meet the specifications imposed on them in previous activities. Validation is an attempt to ensure that the right product is built, that is, the product fulfills its specific intended purpose.

V&V traditionally relies on manual examination of software requirements and design artifacts and the systematic or random testing of target code. As software-intensive systems become increasingly complex, traditional V&V techniques are inadequate for locating the subtle errors in the software.

Claims have been made that the use of formal methods will help improve the quality of software [3, 4]. Formal Validation & Verification (FV&V) of reactive systems has received considerable academic attention during the last three decades. Nevertheless,

[1] A reactive system is a system that changes its actions, outputs and conditions/status in response to stimuli from within or outside it. It is an event-driven or control-driven system continuously having to react to external and/or internal stimuli; that is, the system exhibits non-terminating behavior and reaction to stimulus provided by the environment.

FV&V techniques have not been widely adopted by industry or government even for use in safety-critical commercial and defense applications. For example, although NASA has heavily invested in FV&V research and development, the agency has not adopted FV&V techniques beyond sporadic, almost anecdotal, experimental trials [5-11].

There are numerous possible explanations for this lackluster practical acceptance of FV&V techniques. Clearly, the absence of an ideal technique that can demonstrate life and cost savings has not helped. A more fundamental problem is that software development is a multi-facet process. Each phase of this process has its unique set of problems and there will never be a one-size-fits-all solution for all software development problems. There is a lack of a clear and common understanding about the effectiveness of the spectrum of formal FV&V techniques in different phases of the software development process. So, how can one select the right tool for the right job in FV&V?

In this article, we present a visual tradeoff space, called the FV&V *tradeoff cuboid*, for software engineers to discuss the various tradeoffs (e.g. cost, coverage, etc.) between different FV&V approaches in order to select the appropriate techniques for V&V. We illustrate the use of the tradeoff space with a discussion of cost and coverage tradeoffs among three categories of FV&V techniques: theorem proving, non-execution-based model checking, and execution-based model checking via the combination of runtime verification and automatic test generation. We show, using the cuboid, the pros and cons of the three categories of techniques.

2. THE V&V REQUIREMENTS IN THE SOFTWARE LIFE CYCLE

One can view software development as a set of transformations via the following workflows: *requirements specification*, *design*, and *implementation*. Depending on the software process model, these transformations may be carried out in a sequential order (as in the Waterfall, or Spiral processes), or in an iterative and incremental fashion (as in the Unified process). Table 1 shows the input/output of each transformation and the corresponding V&V activities.

Development Activities	**Input**	**Output**	**V&V Activities**	
Requirements Specification	Clients' ideas	System/software functional and non-functional requirements	Assure the adequacy, correctness, and consistency of requirements; develop acceptance test plan and test cases	Validation
Design	System/software requirements	Architecture/ component specification	Assure the consistency of design with requirements, and the adequacy of design; develop integration and unit test plan and test cases	Verification
Implementation	Architecture/ Component specification	Target Code	Assure the consistency of code with design, and the adequacy of the implementation, execute the tests as planned	

Table 1. The Life-cycle V&V activities.

Clark *et al* reported in [3] that the process of specifying requirements formally enables developers to gain "a deeper understanding of the system being specified," and to "uncover requirements flaws, inconsistencies, ambiguities and incompletenesses." In addition, the artifacts produced by enacting the process "can itself be formally analyzed," thus allowing the possibility for some degree of automation of V&V tasks.

In [12], Berry pointed out that most errors, between 65% and 85%, are introduced into the software-intensive reactive systems "during the requirements discovery, specification, and documentation stages," and only about 25% of the errors are introduced during the coding stage. Hence, it is most cost effective to apply formal methods on requirements validation. Berry further illustrated his point with the following figure at the 1998 Monterey Workshop on Engineering Automation for Computer-Based Systems.

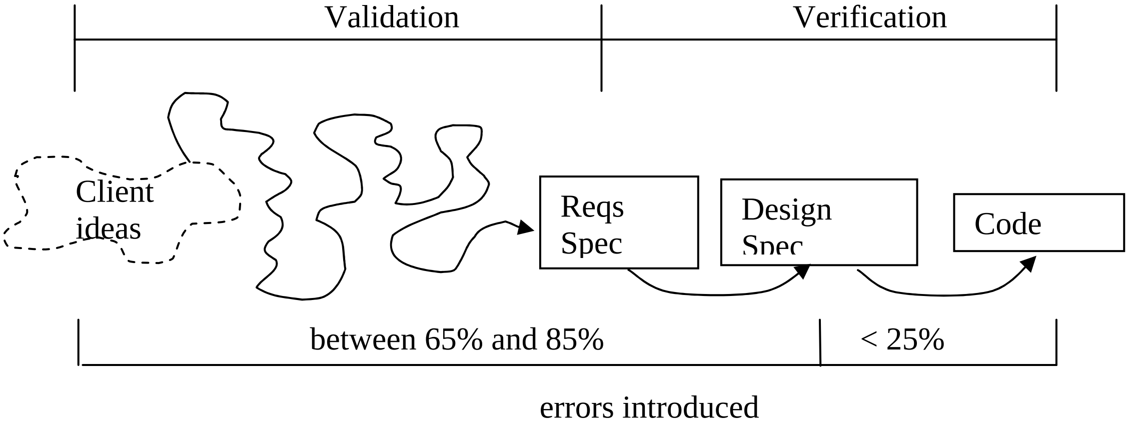

Figure 1. The perils of requirement analysis

The wavy line between "client ideas" and "Reqs Spec" in Figure 1 represents the inherent uncertainty and difficulty in nailing down the correct requirements, while the relatively "smooth" arrows from "Reqs Spec" to "Design Spec" and from "Design Spec" to "Code" indicate the potential for systematic (and possibly mechanical) transformations toward the target system once we have the correct requirements specifications. Figure 1 also highlights the iterative and incremental nature of the validation process.

Moreover, as Lutz pointed out is her study of the software errors discovered during the integration and testing phase of the Voyager and Galileo spacecraft, the majority of the program faults were functional faults, and a large percentage of the functional faults were behavioral faults (50% of the safety-related, functional faults in Voyager and 38% of safety-related, functional faults in Galileo) [13]. Lutz's finding highlights the difficulties in understanding and implementing behavioral requirements correctly. Hence, it pays to invest in FV&V techniques that help validate behavioral requirements and detect behavioral errors.

We need to separate the FV&V techniques into two categories: the FV&V for the Requirements phase and the FV&V for the Design/Code phase. The FV&V techniques for the Requirements phase are formal *validation* techniques. These techniques must allow stakeholders to capture the formal requirements (e.g. via simulations) to assure that the developer's cognitive understanding of the requirements matches the formal specifications. The FV&V techniques for the Design/Code phase are formal *verification* techniques. These techniques should allow developers to achieve the level of confidence that their software satisfies the requirements (functional and non-functional), and should effectively locate and explain the cause of errors in faulty design and code.

3. THE FV&V DIMENSIONS

Let us return to our discussion of the dimensions of the FV&V tradeoff space, which is made up of the following three dimensions – *specification/validation*, *program/application*, and *verification*.

3.1 THE SPECIFICATION/VALIDATION DIMENSION

The *specification/validation dimension* represents the cost, effort and effectiveness associated with formal specification. Formal requirements specification is the process of capturing requirements and properties for the domain of discourse (component, module, or system being designed or inspected) in a machine interpretable or executable form. The formal specifications describe what *any* system that solves the real-world problem ought to do.

The specification/validation dimension deals with the ease of writing formal specifications and getting them right, that is, getting them to represent the cognitive intent the human owner has or had for this requirement. This dimension measures cost and coverage. Cost is the fiscal cost of creating and validating correct representative formal specifications for desired properties. Coverage is the degree to which a given specification language can actually be used to capture certain properties; a weak formal specification language can only capture simple requirements. For example, the specification language known as Propositional Linear-time Temporal Logic (PLTL) is known to be star-free regular [14] and cannot therefore formally capture requirements that require a stronger formalism, such as requirements that require nontrivial counting. In addition PLTL cannot be used to capture requirements that contain real-time constraints.

3.1.1 Assertion-based Specification vs. Model-based Specification

We classify formal behavioral requirements specifications into two categories – *assertion-based specifications* and *model-based specifications*.

With *assertion-based* specifications, high-level requirements are decomposed into more precise lower-level requirements that are mapped one-to-one to formal assertions. For example, we may start with a high-level requirement:

R1. The system shall not exceed 75% of its maximum load capacity at runtime.

and derive the lower-level requirement:

R1.1 Whenever the system load (L) exceeds 75% of the MaxLoad,
L must be reduced back to 50% of the MaxLoad within 1 minute and must remain at or below 50% of the MaxLoad for at least 10 minutes.

The requirement R1.1 will, in turn, be mapped to a formal assertion expressed either as a Metric Temporal Logic (MTL) [36] assertion:

$$\begin{aligned}&\mathit{Always}\ (L >= 0.75 * \mathit{MaxLoad}\ \mathit{Implies}\\ &\quad (\mathit{Eventually}_{<=1}\ \mathit{min}\ \ (\mathit{Always}_{<=10}\ \mathit{min}\ L <= 0.5 * \mathit{MaxLoad})))\end{aligned}$$

or as a Statechart assertion [39] shown in Figure 2.

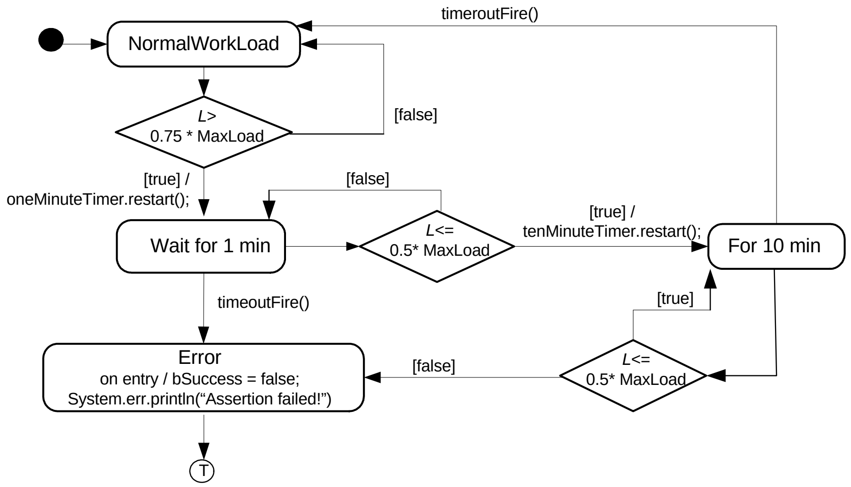

Figure 2. A sample Statechart assertion

With *model-based* behavioral specifications, a *single* monolithic formal model (either as a state-based system or an algebraic-based system) is created to capture the combined expected behavior described by the lower-level requirements. Note that this formal model describes the expected behavior of a conceptualized system from the Requirement space. It may differ significantly from the models derived from the system in the Design/Code space.

This paper is concerned with, and also advocates, the assertion-based specification approach. Its advantages over the model-based specification approach are the following:

1. Requirements are written by humans and need to be traceable in the formal specification. Requirements are indeed traceable in the assertion-based formal specification approach because they are represented, one-to-one, by assertions (acting as watchdogs for the requirements).

A monolithic model specification on the other hand is the sum of all concerns. Hence, upon detecting a violation of the formal specification it is difficult to map that violation to a specific human-driven requirement.

2. When a requirement changes, it is harder to adjust the monolithic model without affecting behavior related to other requirements. Hence, assertion-based specifications have a much lower maintenance cost than the model-based counterpart.
3. Particular assertions can be constructed to represents illegal behaviors, whereas in the monolithic model approach the formal model typically only represents "good behavior."
4. It is much easier to trace the expected/actual behaviors of the target system to the required behaviors in the Requirements space with assertion-based specifications than with the model-based specifications. The requirements assertions can be used directly as input to the verifiers in the verification dimension.
5. The conjunction of all the assertions becomes a "single" formal model of a conceptualized system from the Requirement space, and can be used to check for consistency and conflicts in the specifications with the help of computer-aided tools.

3.2 THE PROGRAM/APPLICATION DIMENSION

The *program/application dimension* deals with the ease of the adaptation of a given real-life complex application to a specific FV&V technique. In an ideal world we could use an existing application verbatim for our FV&V technique of choice. In reality however this is almost never the case, and an application needs to be modified, truncated, or simplified to be considered for FV&V. For example, a model checker such as SPIN [30] cannot be used verbatim on a non-trivial C, C++, or Java application; rather, such an application needs to go through a process of abstraction before it can be used for verification, and hence has a low program coverage and a high program cost.

3.3 THE VERIFICATION DIMENSION

The *verification dimension* is the dimension that bridges the specification and application dimensions. Verification ensures that the application conforms to the specification. Formal verification does so using computer-based techniques and therefore requires formal specifications for the requirements as well as an executable target system. The verification dimension represents the cost, effort, and effectiveness of verification. For example, it is generally accepted that manual (i.e., human-based) testing is costly, slow, and error prone; it will therefore be represented as a point whose verification dimension highlights the high-cost and low-coverage of manually conducting software testing.

4. QUALITATIVE COMPARISON OF FV&V TECHNIQUES FOR REACTIVE SYSTEM BEHAVIORS

The coverage cuboid, shown in Figure 3, represents the coverage-space tradeoff between three FV&V techniques. Each point in the solid represents the extent of coverage in each dimension provided by a given FV&V technique. Hence, an FV&V technique with high coverage (e.g., high specification coverage) is *better* in that aspect than a technique with low coverage.

Figure 4 is the cost cuboid; it represents the cost-space tradeoff between the three FV&V techniques. Each point in the solid represents the cost in each dimension induced by a given FV&V technique. Clearly, an FV&V technique with high cost (e.g., high verification cost) is *worse* in that aspect than a technique with a low cost.

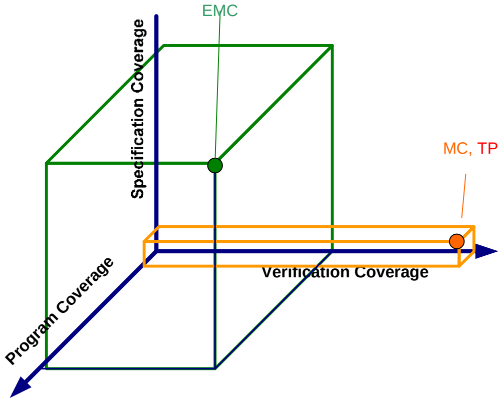

Figure 3. The coverage space

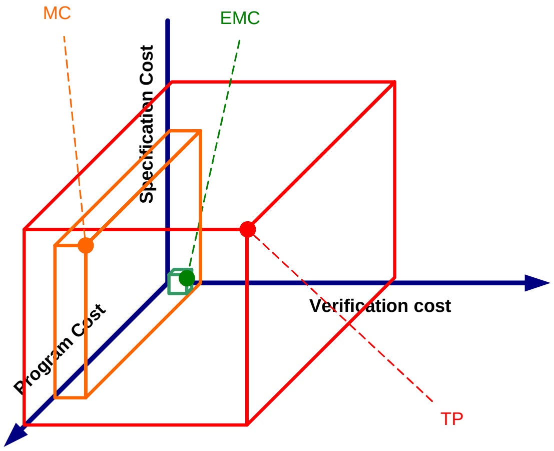

Figure 4. The cost space

4.1 THEOREM PROVING

As it name suggests, *Theorem Proving* (TP) is a formal verification technique that uses mathematical techniques to make a convincing argument that a program conforms to a formal requirement. FV&V TP's always require a human driver because the underlying problem they are trying to solve is typically undecidable. In addition, the choice of the specification language affects the skill level required by the driver. For example, ACL2 [15] uses Propositional-Logic (PL) specification, a Lisp programming style notation for specification, whereas STeP (the Stanford Temporal Prover) [16] uses Propositional Linear-time Temporal Logic (PLTL) for specification [17], a language that requires more expertise than PL. HOL theorem provers [18] are a family of interactive theorem proving systems that use first order logic, which is theoretically as descriptive as PLTL but is arguably harder to use when it comes to specification of reactive system requirements. Examples of HOL TPs include the NQTHM theorem prover [19], HOL4 [18], Isabelle [20], ProofPower [21] and PVS [22], and there were several efforts to embed temporal logic in HOL [23-26]. In addition, there are a number of formal methods that can be used during the code development phase to allow the verification of target code via TP. For example,

1. **Using Floyd-Hoare Logic** [27,28]**:** In this method, every programming step has a pre-condition, post-condition and an invariant. The verifier is expected to use a proof system and check that the post condition follows from the precondition while the invariant is valid.
2. **Using Type systems** [29]**:** The verification and validation can be moved to the design stage by formally stating the requirements in constructive Logic. The programmer, then acts as a mathematician and proves that the requirement is a theorem that follows from the domain axioms. The system then extracts the code automatically from this proof. Therefore the generated code now automatically becomes correct, as the programmer indirectly proved it to be so.

The specification/validation dimension of TP. In order to overcome both the undecidable and the intractable nature of the formal logic systems, it is necessary to limit the expressiveness of the specification languages in order to have practical TP techniques. In general, the more powerful the theorem prover, the more restrictive is its specification language. Existing theorem provers have rather weak and hard-to-use text-based specification languages (mostly based on some form of temporal logic). In contrast, it is a common practice for system designers to model and program using visual languages. We believe the same motivation applies to formal specification. It is difficult for system designers who have a limited knowledge of formal logic to visualize the subtle meaning of temporal logic statements in order to validate the correctness of the formal specifications. Consequently, we ranked TP techniques as having low specification coverage and high specification cost.

The program/application dimension of TP. TP techniques work on special programming languages tailored specifically for the TP process. Hence it is not possible to perform TP on an existing Java or C++ application verbatim. In other words, an existing complex application needs to be first translated into a new representation using the TP tool's language of choice. In most safety-critical application, such as NASA flight-code, or complex defense applications (e.g., the AEGIS weapon system), the new representation will not cover all aspects of the original program; for example, STeP does not have nearly the same library support as Java or C++. Consequently, we ranked TP techniques as having low program coverage and high program cost.

The verification dimension of TP. As discussed above, TP is never automatic, and requires a high level of expertise on the part of the user in automated reasoning. Even with such expertise, it is not guaranteed that the TP process will be completed because of the undecidability of the formal logic systems. Nevertheless, when the process does complete it provides 100% coverage, that is, no more testing is required for that specific specification requirement. Hence, we ranked TP techniques as having good verification coverage but high verification cost.

4.2 MODEL CHECKING

Classical, or non-execution-based, *Model Checking* (MC) is an algorithmic formal verification technique. MC is a push-button verification technique in that once a program is set-up for MC and a property (e.g., reachability, safety, liveness, and fairness[2]) is formally captured using the formal specification language of choice, the process requires no sophisticated driver.

The specification/validation dimensions of MC. Contemporary MC techniques are limited in the specification dimension. For example, SPIN [30] uses PLTL or Büchi-automata for requirement specification, resulting is the similar specification coverage and cost limitations as TP techniques. Kronos [31] and Uppall [32], on the other hand, use timed automata to verify real-time properties specified in computation tree logic (CTL) [33]. Both CTL and PLTL are rather weak subsets of full branching time logic (CTL*) [34]. Both CTL and CTL* use path operators, making it challenging to formulate correct specifications with recursion. Like the formal specifications in the TP techniques, specifications for the MC techniques are text based and difficult to visualize and validate by system designers. Unlike TP, MC does not require the detailed assertions (e.g. invariants) to help guide the intermediate steps of the proof processes. Hence, we rank MC as having low specification coverage and a specification cost slightly lower than the TP's.

The program/application dimension of MC. Model checking's greatest limitation is typically considered to be the *state-space explosion problem*, where the size of the problem space as seen by the MC grows exponentially as the program under verification grows. Consequently, MC is limited to finite-state components and is performance-constrained by the number of states in that component. For example, a single 32-bit integer variable induces effectively 2^{32} states. Consequently, for FV&V of large real-life systems there are two options available for MC users: (i) to ignore large parts of the system using a process known as *abstraction* [35], where MC is performed on a small abstract model of the original system, (ii) to carve out limited, small, parts of the system and perform MC only on those parts. In either case there is a non-trivial effort involved, we therefore rank MC as having high program cost. In addition, the artifact that is eventually model-checked differs significantly from the original system, being either an abstract version or limited portion of the original system. We therefore rank MC as having low program coverage and high program cost.

The verification dimension of MC. The premise of MC is automatic, "push-button" verification, no special driver required. Also, there is 100% verification coverage of the component being verified, if that component is not large. Hence, we rank MC as having high verification coverage and low verification cost.

4.3 EXECUTION-BASED MODEL CHECKING

Runtime Verification (RV) is a verification technique that monitors the runtime execution of a system and checks the observed runtime behavior against the system's formal specification. Hence, RV behaves as an automated observer that observes the

[2] Reachability refers to the condition that certain states are part of a run. Safety refers to behavior that does not (or must not) happen. Liveness refers to conditions like If *x* happens, then *y* must happen. Fairness refers to condition that certain states should be part of every run.

program's behavior and compares it with the expected behavior per the formal specification.

Some RV tools are the TemporalRover/DBRover [36], PaX[37] and RT-Mac [38] that use extensions and variants of PLTL as the specification of choice, and the StateRover [39] that uses deterministic and non-deterministic statechart diagrams as its specification language.

Execution-based Model Checking (EMC) is a combination of RV and Automatic Test Generation (ATG). With EMC, a large volume of automatically generated tests are used to exercise the program or system under test (SUT), using RV on the other end to check the SUT's conformance to the formal specification.

Some ATG tools that, when combined with RV tools, create an EMC technique are the StateRover's white-box automatic test-generator [40] and NASA's Java Path Finder (JPF) [41].

The specification/validation dimension of EMC. Although some early RV tools have used limited specification languages such as PLTL [17] and MTL [42], there is nothing inherent in the ATG, RV, and EMC techniques that limit the specification language. Indeed, the StateRover's specification language is Turing equivalent. In contrast, no specification language for MC or TP is Turing equivalent. In addition, the current state-of-practice considers UML diagrams as easy to use modeling and specification languages, rendering UML-based formal specification less costly to perform and more powerful than specification languages used by MC and TP techniques. The availability of executable code for the formal assertions allows system designers to test specifications (via scenario simulation) independent of the prototype design, ensuring that the system designers truly understand the required system behavior without being tainted by any pre-conceived solutions [43]. Hence, we rank EMC as having high specification coverage and low specification cost.

The program/application dimension of EMC. The premise of RV is that it can be used for FV&V of any existing, unmodified Java, C, or C++ system, regardless of its size and complexity. We therefore rank EMC as having high program coverage and low program cost.

The verification dimension of EMC. EMC is an execution-based FV&V method - both the system under test and the specification are executed in tandem. Consequently, there is always a possibility that the ATG did not generate a test sequence that violates a requirement. Hence EMC's verification coverage cannot be 100% and we therefore rank EMC as having lower verification coverage then MC or TP. Depending on the level of automation of the test-generator, EMC is fully or partially automatic. EMC has a low verification cost when using an automatic ATG tool.

5. CONCLUSION

Clearly, as visually depicted by Figures 3 and 4 there exists no ideal FV&V technique. Hence, an organization may need to determine how to best allocate the limited resources it has to fulfill these activities. For example, an organization that chooses TP or MC is effectively deciding to favor good verification but for a restricted set of behavioral (reactive) requirements, since many behavioral requirements of interest cannot be addressed by MC. In addition, a choice of MC will limit the size or detail level of the application being verified. EMC on the other hand, when compared with MC and TP, has better specification coverage and cost and better program coverage and cost, but inferior verification coverage.

Consequently, one can conclude from Figures 3 and 4 that the choice boils down to the choice between to:

1. Thoroughly verify a limited application against a limited set of requirements with a high upfront cost of specification-development and program-adaptation.
2. Partially verify an entire application as-is, against a wide set of real-life requirements.

This choice might also help explain the lackluster acceptance of FV&V techniques by the industry. In the past, MC and TP have been the prominent available FV&V techniques, forcing the marketplace to fund verification of limited components against limited, often seen as over simplified, requirements. This was not considered as a good investment for many in the marketplace.

Studies of software failures typically point to the importance of correct requirements and the difficulties in getting the correct description of these requirements. One must start with the correct requirements specifications. Otherwise, it does not matter how effective and efficient a verification technique is; it is an exercise in futility to formally verify that a system behaves “correctly” according to invalid requirements (i.e., built the wrong system). Hence, it is important to select the FV&V techniques that are both cost-effective and coverage-effective in the specification/validation dimension.

We advocate the assertion-based over the model-based approach to V&V for requirements specifications because the former allows the system developers to modularize their thinking and focus on each property (or sets of properties) in isolation. In additions, it is much easier to verify the behavior of the actual system against each assertion (or sets of assertions) than comparing the equivalence of two monolithic formal models.

6. ACKNOWLEDGEMENT

The research was funded in part by a grant from the National Aeronautics and Space Administration. The views and conclusions in this talk are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. We thank Duminda Wijesekera and Butch Caffall for reviewing this report.

7. REFERENCES

1. RTI, *The Economic Impacts of Inadequate Infrastructure for Software Testing*, Planning Report 02-3, National Institute of Standard and Technology, May 2002. http://www.nist.gov/director/prog-ofc/report02-3.pdf.

2. P. Bourque and R. Dupuis, eds., S*webok: Guide to the Software Engineering Body of Knowledge (2004 Version)*, IEEE, 2004.

3. E. Clarke, J. Wing, et. al., "Formal Methods: State of the Art and Future Direction," *ACM Computing Surveys*, vol. 28, no. 4, Dec. 1996, pp. 626-643.

4. S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton, "Experiences Using Lightweight Formal Methods for Requirements Modeling," *IEEE Trans. Software Eng.*, vol. 24, no. 1, Jan. 1998, pp. 4-14.

5. R. Butler, J. Caldwell, V. Carreno, C. Holloway, P. Miner, and B. Di Vito, "NASA Langley's Research and Technology-Transfer Program in Formal Methods," *Proc. 10th Annual Conf. Computer Assurance*, IEEE, 1995, pp. 135-149.

6. D. Hamilton, R. Covington and J. Kelly, C. Kirkwood, M. Thomas, A. R. Flora-Holmquist, M. G. Staskauskas, S. P. Miller, M. Srivas, G. Cleland, and D. MacKenzie, "Experiences in Applying Formal Methods to the Analysis of Software and System Requirements," *Proc. 1st Workshop Industrial-Strength Formal Specification Techniques*, IEEE, 1995, pp. 30-43.

7. M. Hinchey, J. Rash, and C. Rouff, "A formal approach to requirements-based programming," *Proc. 12th Int'l Conf. Engineering of Computer-Based Systems*, IEEE, 2005, pp. 339–345.

8. H. Holt, "Assessment of Fault-Tolerant Computing Systems at NASA's Langley Research Center," *Proc. IEEE Aerospace Conf.*, vol. 2, IEEE, 1997, pp. 541-549.

9. M. Lowry, M. Boyd, and D. Kulkami, "Towards a theory for integration of mathematical verification and empirical testing," *Proc. 13th IEEE Int'l Conf. Automated Software Engineering*, IEEE, 1998, pp. 322-331.

10. J. Rash, M. Hinchey, C. Rouff, and D. Gracanin, "Experiences with a Requirements-based Programming Approach to the Development of a NASA Autonomous Ground Control System," *Proc. IEEE Workshop Engineering Autonomic Systems*, IEEE, 2005, pp. 490-497.

11. C. Rouff, A. Vanderbilt, W. Truskowski, J. Rash, and M. Hinchey, "Verification of NASA emergent systems," *Proc. 9th IEEE Int'l Conf. Engineering Complex Computer Systems*, IEEE, 2004, pp. 231-238.

12. D. Berry, "Formal Methods: The Very Idea, Some Thoughts About Why They Work When They Work," *Electronic Notes in Theoretical Computer Science*, vol. 25, 1999, http://www.elsevier.nl/locate/entcs/volume25.html.

13. R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," *Proc. IEEE Int'l Symp. Requirements Engineering*, IEEE, 1993, pp. 26-133.

14. J. Cohen, D. Perrin and J.-E. Pin, "On the Expressive Power of Temporal Logic," *J. Computer and System Sciences*, vol. 46, no. 3, 1993, pp. 271-294.

15. M. Kaufmann, P. Manolios, and J. S. Moore, *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, 2000.

16. N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe, "STeP: Deductive-Algorithmic Verification of Reactive and Real-time Systems," *Proc. 8th Int'l Conf. Computer Aided Verification*, LNCS 1102, Springer-Verlag, 1996, pp. 415-418.

17. U. Nitsche, "Propositional Linear Temporal Logic and Language Homomorphisms," *Proc. 3rd Int'l Symp. Logical Foundations Computer Science*, LNCS 813, Springer-Verlag, pp. 265-277.

18. M. J. C. Gordon and T. F. Melham, eds., *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, 1993.

19. R.S. Boyer and J.S. Moore, *A Computational Logic Handbook*, Academic Press, 1988.

20. L. C. Paulson, *Isabelle: A Generic Theorem Prover*, LNCS 828, Springer, 1994.

21. D. King and R. Arthan, "Development of Practical Verification Tools," *ICL Systems J.*, vol. 11, no. 1, 1996, pp. 106-122.

22. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert, *PVS System Guide*, Computer Science Laboratory, SRI International, Menlo Park, Calif., Sept. 1999.

23. R.W. S. Hale, *Programming in Temporal Logic*, Ph.D. thesis, published as technical report 173, Computer Laboratory, Cambridge University, Cambridge, U.K., Oct. 1989.

24. J. J. Joyce, *Multi Level Verification of Microprocessor-Based Systems*, Ph.D. thesis, published as technical report 195, Computer Laboratory, Cambridge University, Cambridge, U.K., May 1990.

25. J. von Wright, "Mechanizing the Temporal Logic of Actions in HOL," *Proc. Int'l Workshop HOL Theorem Proving System and Its Applications*, IEEE, 1991, pp. 155-159.

26. R. Cardell-Oliver, R. Hale and J. Herbert, "An Embedding of Timed Transition Systems in HOL," *Proc. Int'l Workshop Higher Order Logic Theorem Proving and its Applications*, L. J. M. Claesen and M. J. C. Gordon, eds., North-Holland, 1992, pp. 263-278.

27. D. Gries, *The Science of Programming*, Springer-Verlag, New York, 1981.

28. K.R. Apt and E.-R. Olderrog, *Verification and Validation of Sequential and Concurrent Programs* (2nd Ed.), Springer-Verlag New York, 1997.

29. R.L. Constable, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice Hall, New Jersey, 1986..

30. G. Holzmann, "The Model Checker SPIN," *IEEE Trans. Software Engineering*, vol. 23, no. 5, 1997, pp. 279-295.

31. M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: A Model-Checking Tool for Real-Time Systems," *Proc. 10th Int'l Conf. Computer-Aided Verification*, A.J. Hu and M.Y. Vardi, eds., LNCS 1427, Springer-Verlag, 1998, pp. 546-550.

32. K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," *Int'l J. Software Tools for Technology Transfer*, vol. 1, nos. 1-2, 1997, pp. 134-152.

33. E. M. Clarke and E. A. Emerson, "Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic," *Proc. Workshop on Logic of Programs*, D. Kozen, ed., LNCS 131, Springer-Verlag, 1981, pp. 52-71.

34. E. A. Emerson and J. Y. Halpern, "'Sometimes' and 'Not Never' Revisited: On Branching versus Linear Time Temporal Logic," *J. ACM*, vol. 33, no. 1, 1986, pp. 151-178

35. E. Clarke, O. Grumberg, and D. Long, "Model Checking and Abstraction," *ACM Trans. Programming Languages and Systems*, vol. 16, no. 5, 1994, pp. 1512-1542.

36. D. Drusinsky, "The Temporal Rover and the ATG Rover," *Proc. SPIN 2000 Workshop*, LNCS 1885, Springer-Verlag, 2000, pp. 323-329.

37. K. Havelund and G. Rosu, "An Overview of the Runtime Verification Tool Java PathExplorer," *Formal Methods in System Design*, vol. 24, Springer Netherlands, 2004, pp. 189-215.

38. U. Sammapun, I. Lee, and O. Sokolsky, "RT-MaC: Runtime Monitoring and Checking of Quantitative and Probabilistic Properties," *Proc. 11th IEEE Int'l Conf. Embedded and Real-Time Computing Systems and Applications*, IEEE, 2005, pp. 147-153.

39. D. Drusinsky, "Semantics and Runtime Monitoring of TLCharts: Statechart Automata with Temporal Logic Conditioned Transitions," *Proc. 4th Workshop on Runtime Verification*, Electronic Notes in Theoretical Computer Science, vol. 113, Springer, 2005, pp. 3-21.

40. D. Drusinsky, *Modeling and Verification Using UML Statecharts - A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking*, Elsevier, 2006.

41. K. Havelund and T. Pressburger, "Model Checking Java Programs using Java PathFinder," *Int'l J. Software Tools for Technology Transfer*, vol. 2, no. 4, 2000, pp. 366-381.

42. E. Chang, A. Pnueli and Z. Manna, “Compositional Verification of Real-Time Systems,” *Proc. 9th IEEE Symp. Logic in Computer Science*, IEEE, 1994, pp. 458-465.

43. D. Drusinsky, M. Shing, and K. Demir, “Creating and Validating Embedded Assertion Statecharts,” *IEEE Distributed Systems Online*, vol. 8, no. 5, 2007.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library, Code 52
   Naval Postgraduate School
   Monterey, CA

3. Research Office, Code 09
   Naval Postgraduate School
   Monterey, CA

4. Dr. Butch Caffall
   NASA IV&V Facility
   Fairmont, WV

5. LTC Thomas Cook
   Naval Postgraduate School
   Monterey, CA

6. Dr. Doron Drusinsky
   Naval Postgraduate School
   Monterey, CA

7. Dr. Bret Michael
   Naval Postgraduate School
   Monterey, CA

8. Dr. Man-Tak Shing
   Naval Postgraduate School
   Monterey, CA